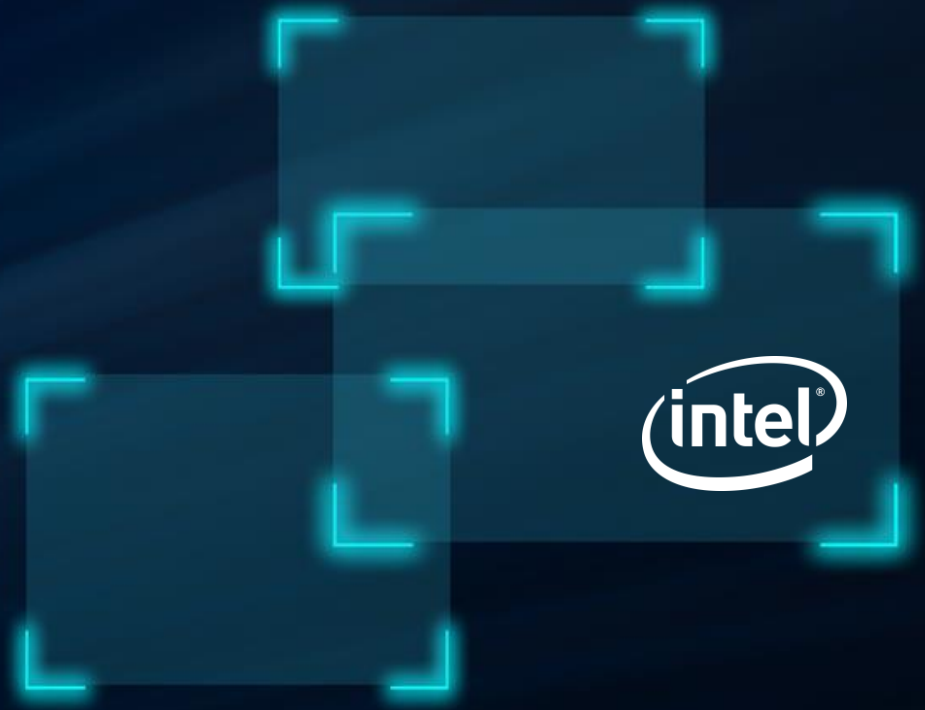
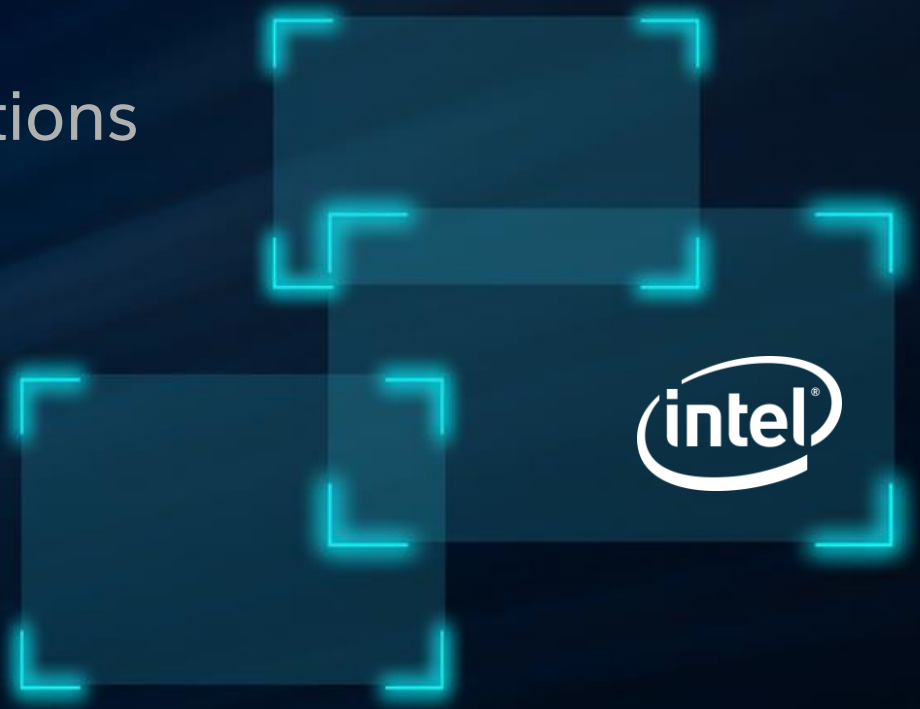


Measuring and Summarizing Latencies Using Trace Events

Tom Zanussi, Intel Open Source Technology Center, ELC 2018

- Trace Events Background
- Latency Calculations and Handlers and Actions
- Latency Histograms and Synthetic Events
- Object-based Latencies
- Using Function Events for Latency Tracing
- Questions



Measuring and Summarizing Latencies Using Trace Events

Tom Zanussi, Intel Open Source Technology Center, ELC 2018

Trace Events Background

- Linux has a large set of ‘trace events’, grouped by subsystem
  - Important places in the kernel where data can be logged to an in-memory buffer

```
root:/sys/kernel/debug/tracing/events# ls
block       filemap    module     timer      cgroup     gpio     hda       napi     sched    kmem
net         scsi       sock       workqueue  drm        i2c      pagemap   i915     power    iommu
synthetic   irq        random     syscall    task       thermal
```

- Every event has a ‘format’ file describing each event field:

```
# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/format
format:
        field:unsigned char common_preempt_count;          offset:3;       size:1;  signed:0;
        field:int common_pid;                              offset:4;       size:4;  signed:1;

        field:unsigned long call_site;                     offset:16;      size:8;  signed:0;
        field:size_t bytes_req;                            offset:32;      size:8;  signed:0;
        field:size_t bytes_alloc;                          offset:40;      size:8;  signed:0;
```

Trace Events Background (cont'd)

- An event or set of events can be 'enabled', which will log the given event(s)
  - The default destination is the ftrace buffer **`/sys/kernel/debug/tracing/trace`**

```
# echo 1 > /sys/kernel/debug/tracing/events/sched/enable

# cat /sys/kernel/debug/tracing/trace

 bash-5141  [004] d...2.. 21721.488735: sched_switch: prev_comm=bash prev_pid=5141
    prev_prio=120 prev_state=D ==> next_comm=swapper/4 next_pid=0 next_prio=120

 <idle>-0     [001] dN..3.. 21721.490873: sched_wakeup: comm=ktimersoftd/1 pid=21 prio=98

 ktimersoftd/1-21    [001] d...2.. 21721.490909: sched_switch: prev_comm=ktimersoftd/1
    prev_pid=21 prev_prio=98 prev_state=S ==> next_comm=swapper/1 next_pid=0 next_prio=120
```

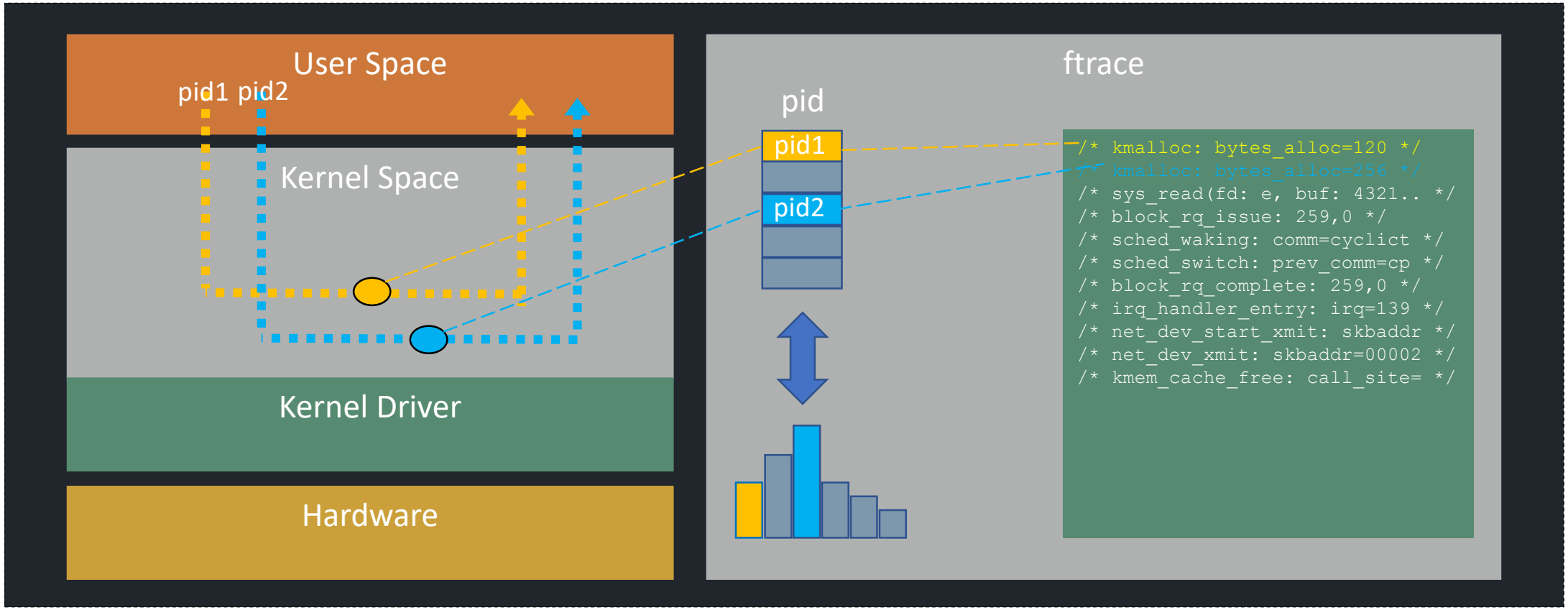
Trace Events Background (cont'd)

- A histogram can be created using event fields as keys and values with the '`hist`' keyword

```
# echo 'hist:keys=common_pid:values=bytes_alloc,hitcount:sort=bytes_alloc.descending' >
        /sys/kernel/debug/tracing/events/kmem/kmalloc/trigger

# cat /sys/kernel/debug/tracing/events/kmem/kmalloc/hist

# event histogram
#

{ common_pid:       3868 } hitcount:        914  bytes_alloc:    2076520
{ common_pid:       1301 } hitcount:        346  bytes_alloc:     708608
{ common_pid:       1270 } hitcount:        186  bytes_alloc:      69152
{ common_pid:       1769 } hitcount:          3  bytes_alloc:       1600
{ common_pid:       1145 } hitcount:          9  bytes_alloc:        576
{ common_pid:       6223 } hitcount:          1  bytes_alloc:         64

Totals:
    Hits: 1912
    Entries: 13
```

Trace Events Background (cont'd)

- A histogram keyed on pid has a bucket for each pid. Each event adds to its pid bucket.



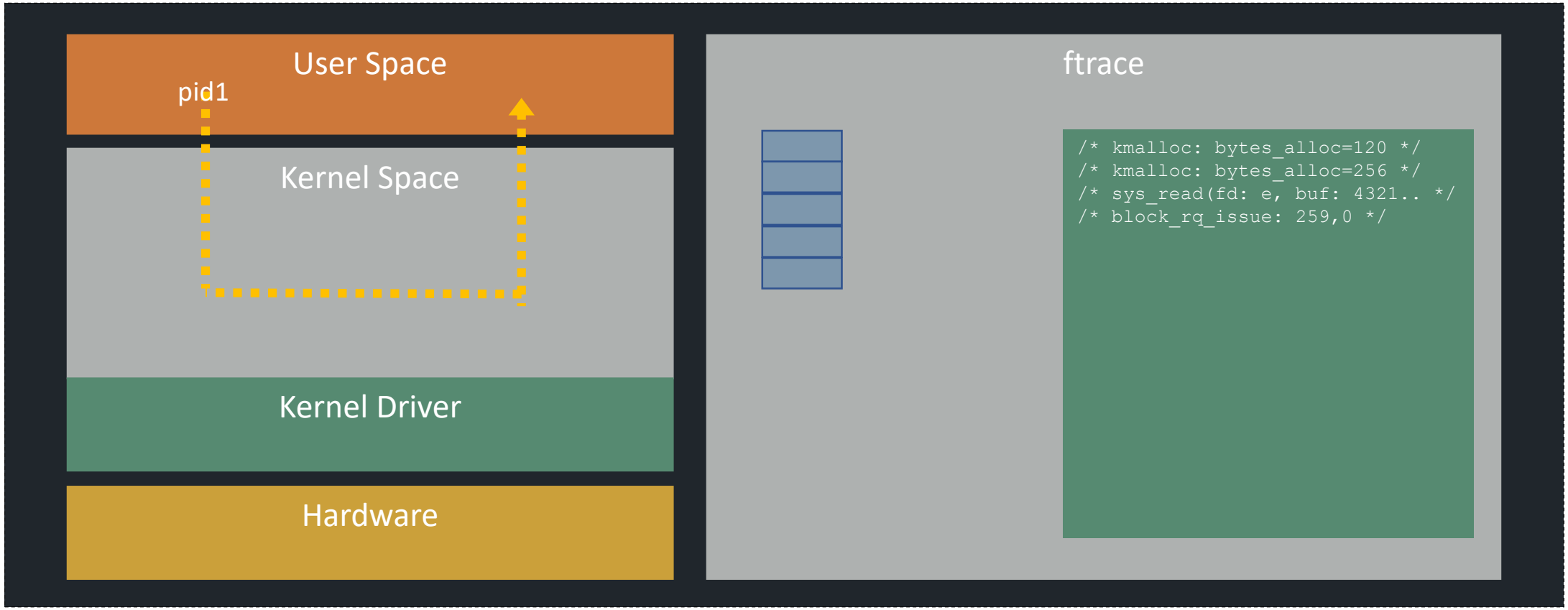
Latency is a derived quantity

- Trigger on two events, save a field value in a variable on the first event
  - Later on in the second event, retrieve that variable and use it

```
# echo 'hist:keys=pid:ts0=common_timestamp.usecs if comm=="cyclictest" && prio < 100'
        >> /sys/kernel/debug/tracing/events/sched/sched_waking/trigger

# echo 'hist:keys=next_pid:waking_lat=common_timestamp.usecs-$ts0 if next_comm=="cyclictest"'
        >> /sys/kernel/debug/tracing/events/sched/sched_switch/trigger

# cyclictest -p 80 -n -s -t 1 -D 2
```

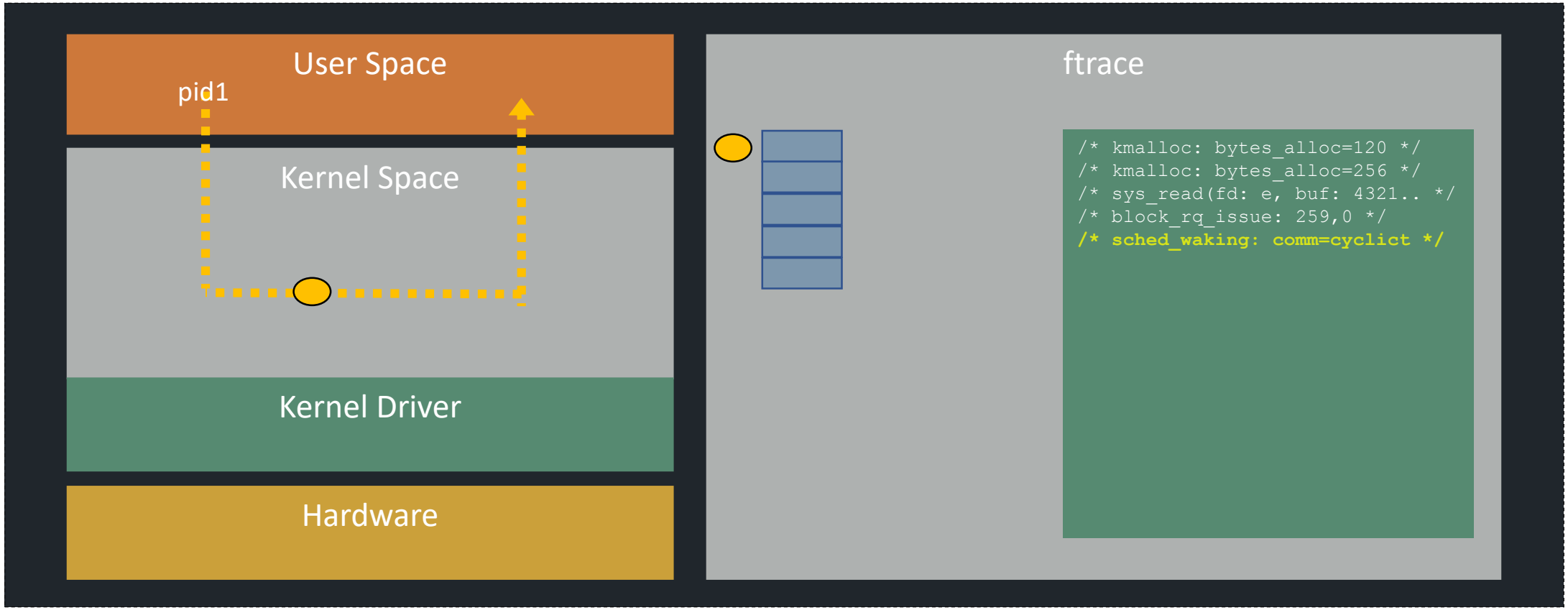
Latency (cont'd)

- A userspace process, say `cyclictest`, makes a system call



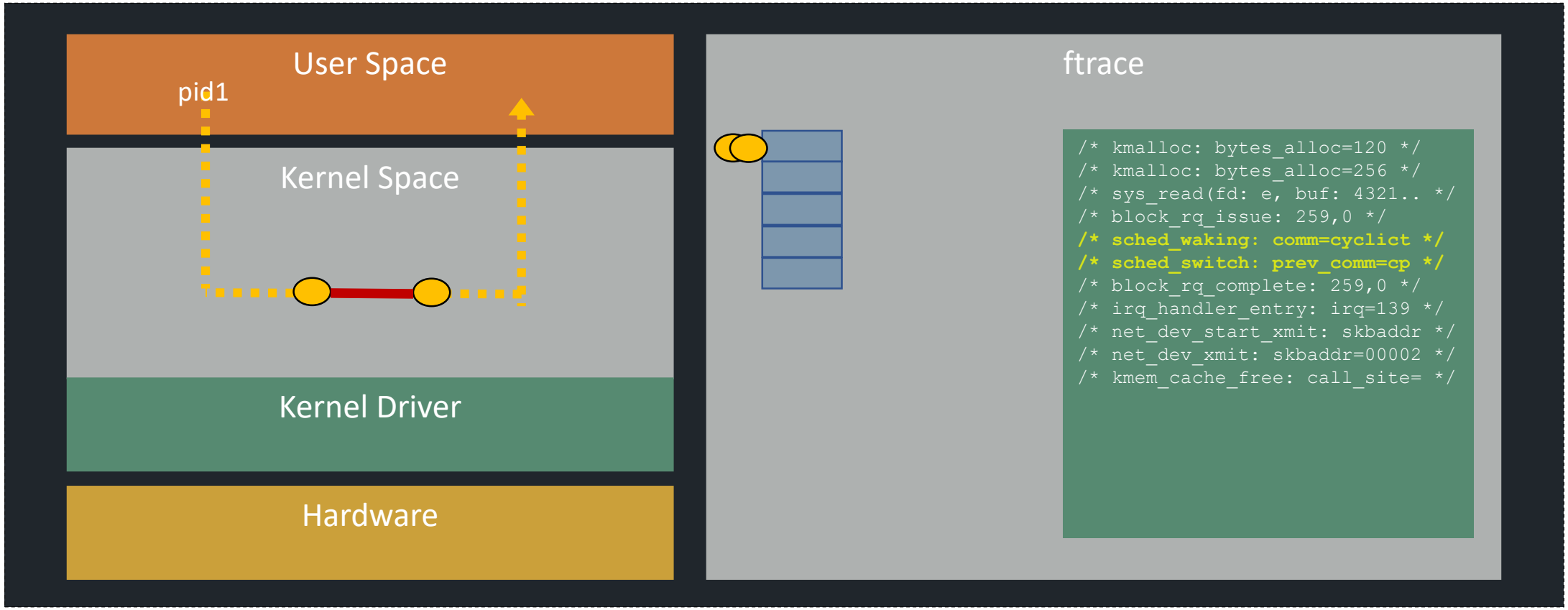
Latency (cont'd)

- The process sleeps and when it awakens the `sched_waking` event happens for that pid



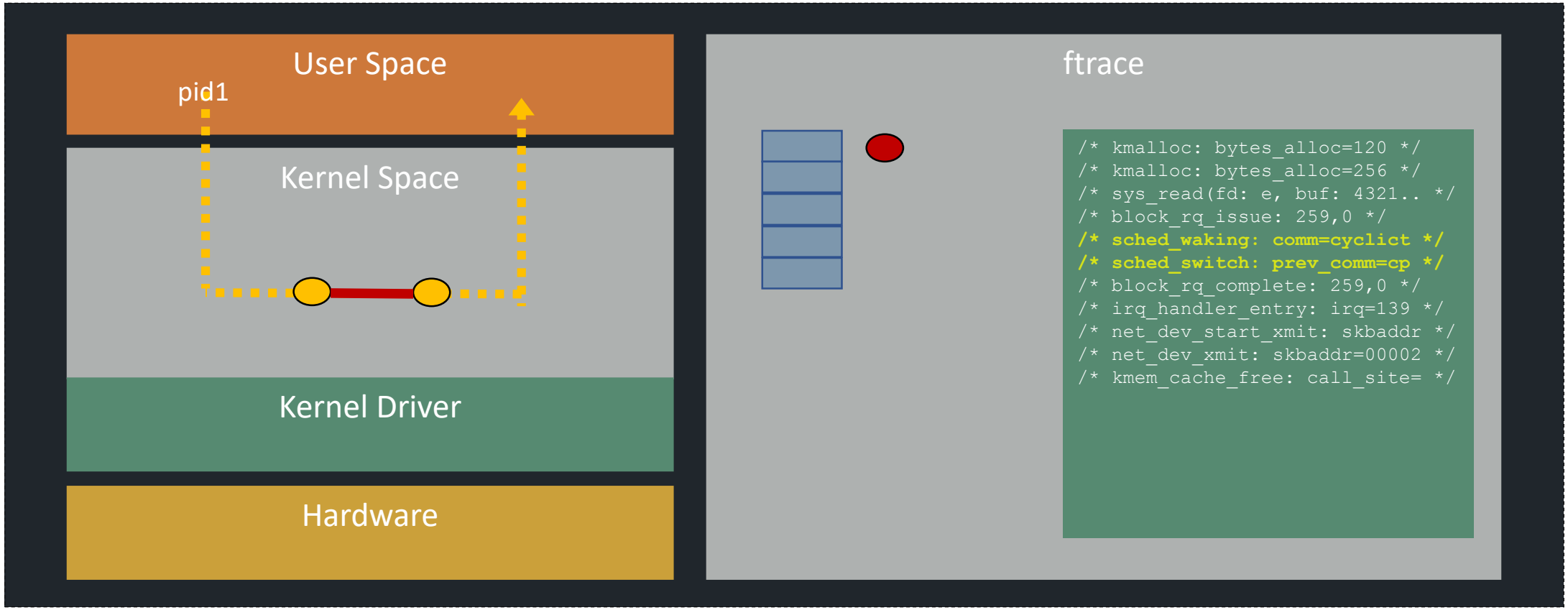
Latency (cont'd)

- When the process is scheduled in the `sched_switch` event is logged



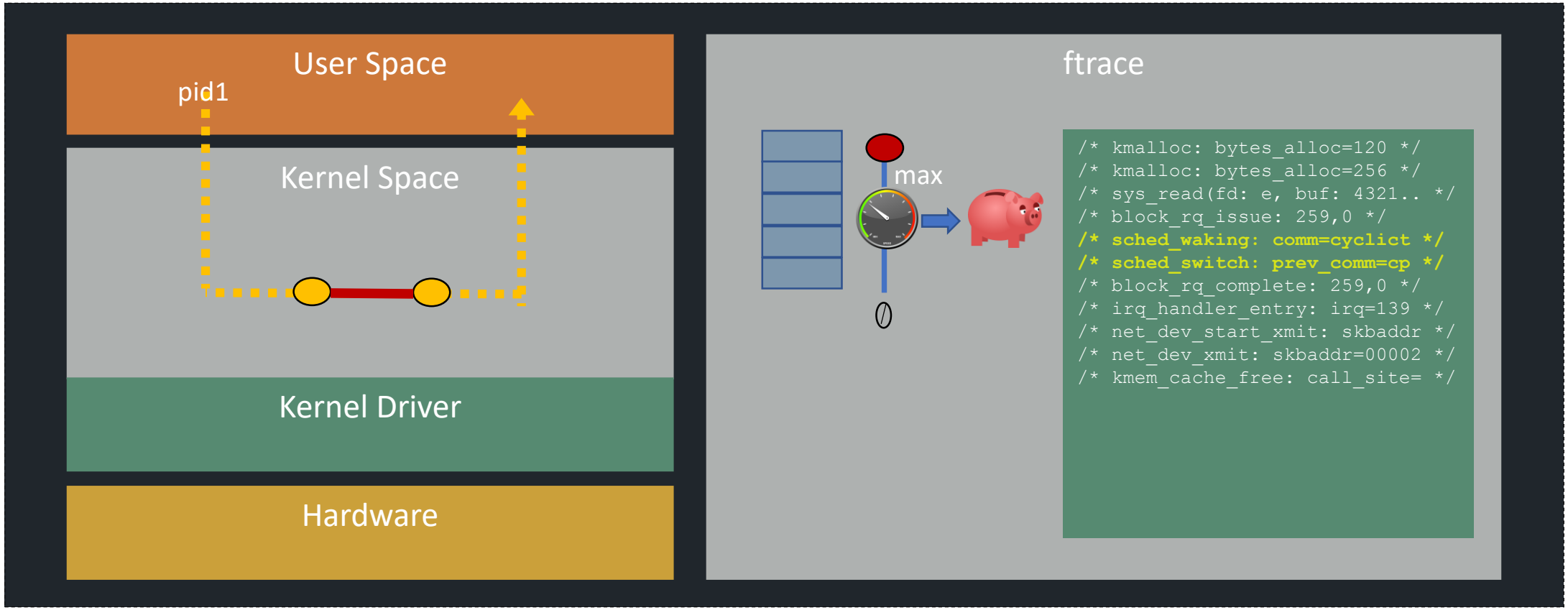
Latency (cont'd)

- The latency value is calculated from the two timestamps, which are discarded after use



We have a latency value, now what?

- We can test if it's the highest latency seen so far and if so, save it and some other things



Handlers and Actions (onmax and save)

- `onmax($variable).save(event fields)`
  - `onmax()` is the 'handler', `save()` is the 'action'

```
# echo 'hist:keys=pid:ts0=common_timestamp.usecs if comm=="cyclictest" && prio < 100'
        >> /sys/kernel/debug/tracing/events/sched/sched_waking/trigger

# echo 'hist:keys=next_pid:waking_lat=common_timestamp.usecs-$ts0:
        onmax($waking_lat).save(next_comm,next_prio,prev_pid,prev_comm,prev_prio)
                if next_comm=="cyclictest"'
        >> /sys/kernel/debug/tracing/events/sched/sched_switch/trigger

# cyclictest -p 80 -n -s -t 1 -D 2
```

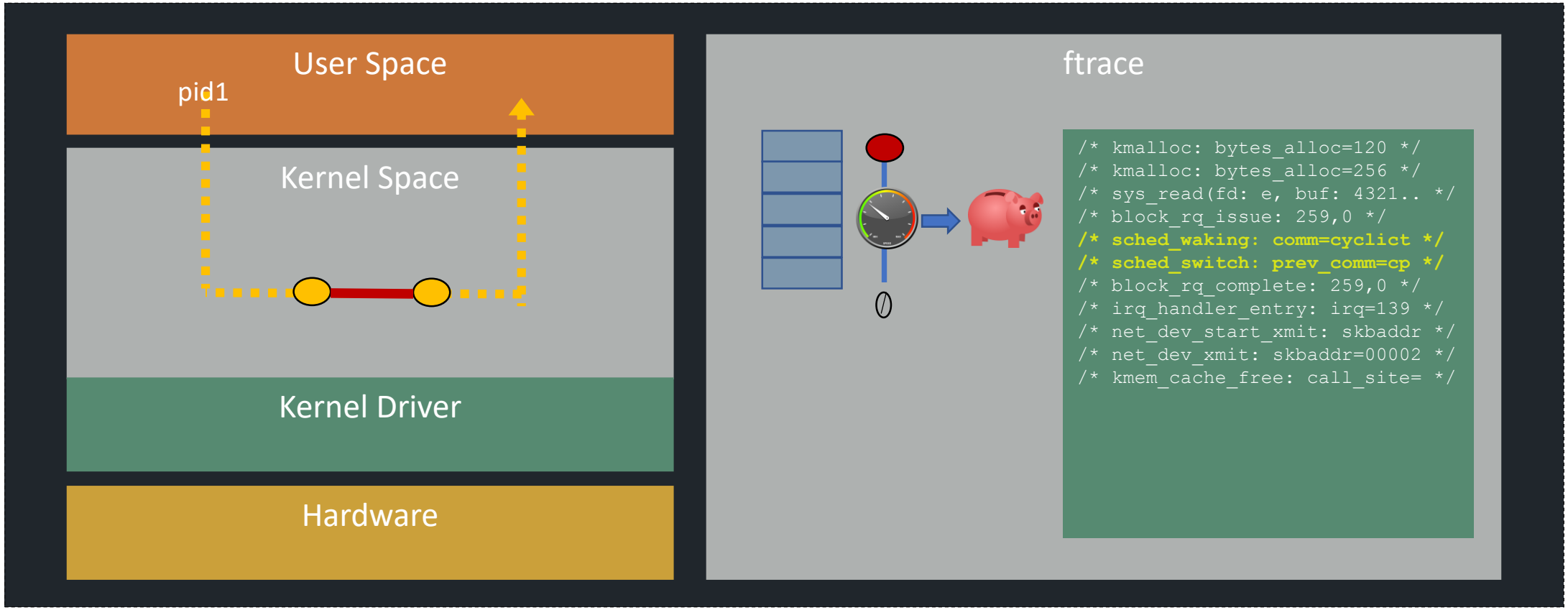
onmax and save output

- We put the `onmax().save()` on the `sched_switch` event's hist trigger
  - Looking at the `sched_switch` hist trigger, we see the saved max and other values

```
# cat /sys/kernel/debug/tracing/events/sched/sched_switch/hist

# event histogram
#

{ next_pid:       1927 } hitcount:       1981
         max:         67  next_comm: cyclictest                    next_prio:         19
         prev_pid:     0  prev_comm: swapper/5                     prev_prio:        120

Totals:
    Hits: 1981
    Entries: 1
    Dropped: 0
    Deletes: 0
```

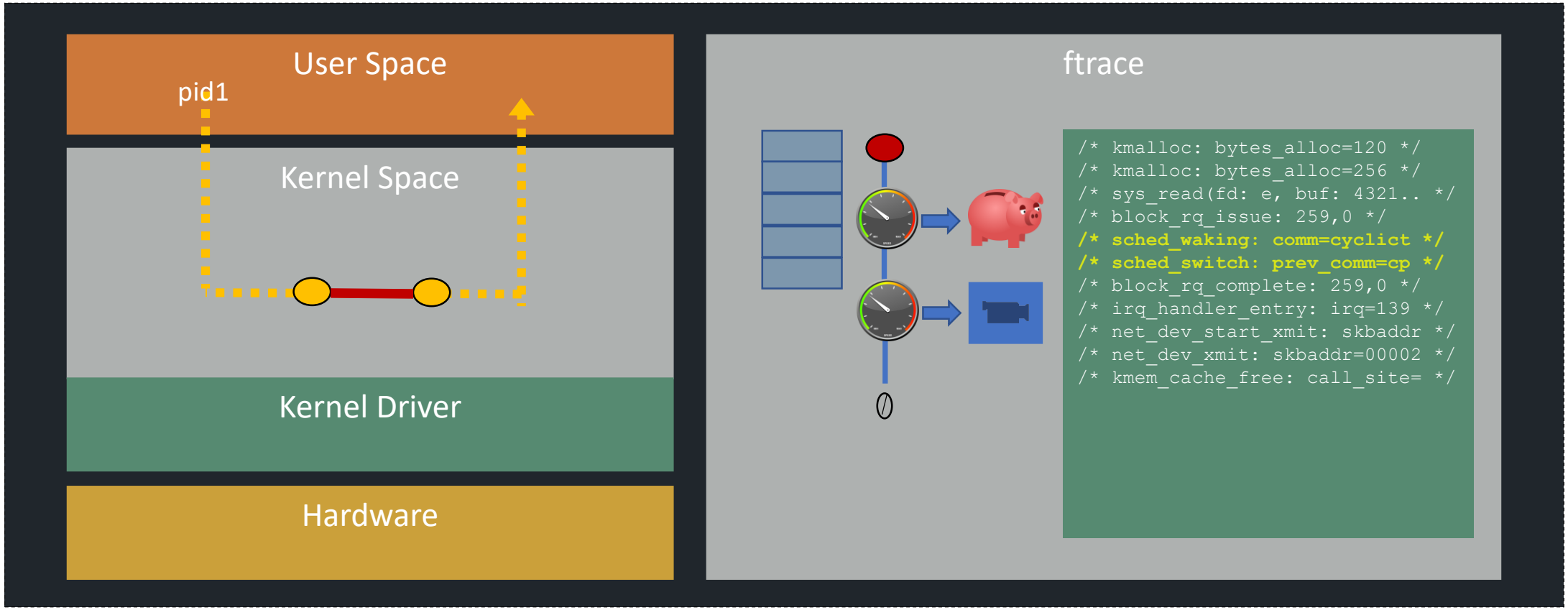
Chaining handlers

- We can add additional handlers and actions, triggered in the order they were defined



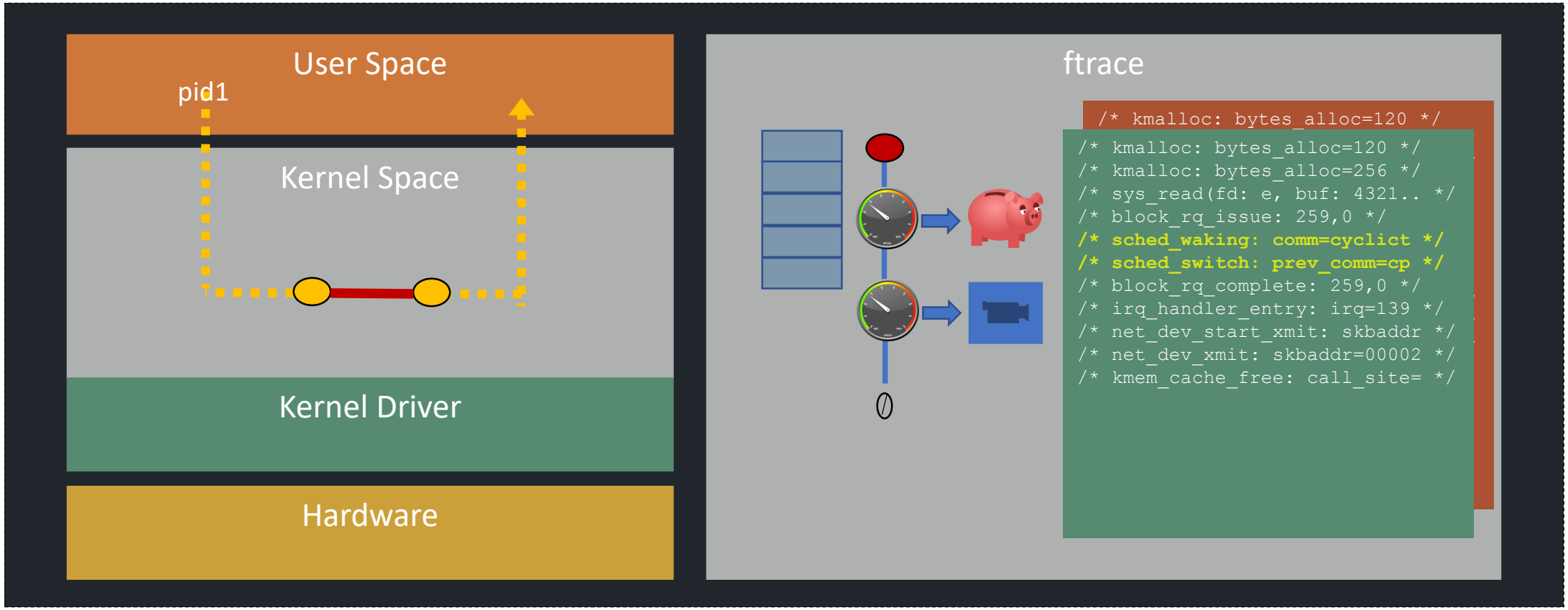
Chaining handlers (add snapshot())

- We can add another action when a new max is hit, like take a snapshot of the trace buffer



Chaining handlers (save + snapshot)

- This allows a capture of all the events that led up to the new max



onmax + snapshot

- `onmax($variable).snapshot()`
  - `onmax()` is the 'handler', `snapshot()` is the 'action'

```
# echo 'hist:keys=pid:ts0=common_timestamp.usecs if comm=="cyclictest" && prio < 100'
        >> /sys/kernel/debug/tracing/events/sched/sched_waking/trigger

# echo 'hist:keys=next_pid:waking_lat=common_timestamp.usecs-$ts0:
        onmax($waking_lat).save(next_comm,next_prio,prev_pid,prev_comm,prev_prio):
        onmax($waking_lat).snapshot()
                if next_comm=="cyclictest"'
        >> /sys/kernel/debug/tracing/events/sched/sched_switch/trigger

# echo 1 > /sys/kernel/debug/tracing/events/enable
# echo 0 > /sys/kernel/debug/tracing/events/lock/enable
# cyclictest -p 80 -n -s -t 1 -D 2
```

onmax and save and snapshot output

- `onmax.save()` output:

```
# cat /sys/kernel/debug/tracing/events/sched/sched_switch/hist

# event histogram
#

{ next_pid:      20049 } hitcount:       1965
         max:         99  next_comm: cyclictest                    next_prio:         19
         prev_pid:     0  prev_comm: swapper/5                     prev_prio:        120
         max:         99

Totals:
    Hits: 1965
    Entries: 1
    Dropped: 0
    Deletes: 0
```

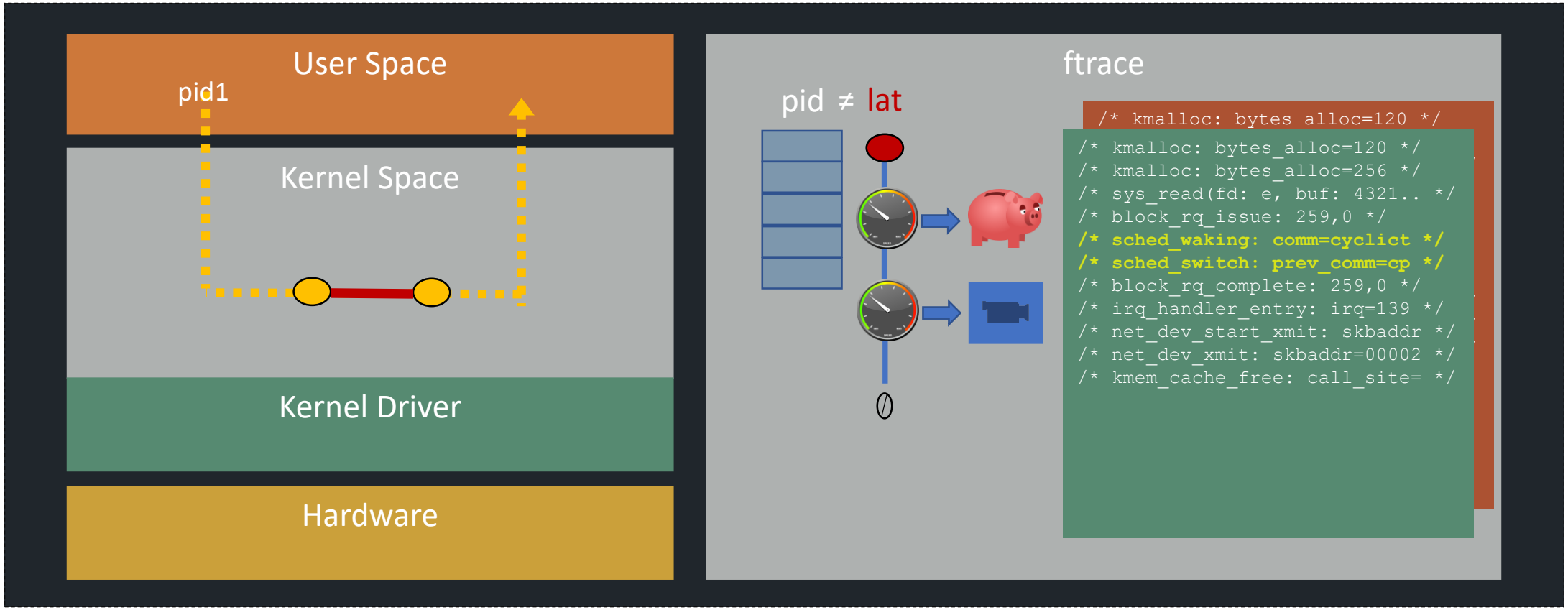
onmax and save and snapshot output

- `onmax.snapshot()` output:

```
# cat /sys/kernel/debug/tracing/snapshot

<idle>-0     [005] d.h2 259727.776377: hrtimer_expire_entry: hrtimer=000000003a968ed8
        function=hrtimer_wakeup now=259728721741149
<idle>-0     [005] d.h3 259727.776383: sched_waking: comm=cyclictest pid=20049 prio=19
<idle>-0     [005] dNh4 259727.776416: sched_wakeup: comm=cyclictest pid=20049 prio=19
<idle>-0     [005] dNh2 259727.776421: hrtimer_expire_exit: hrtimer=000000003a968ed8
<idle>-0     [005] dNh2 259727.776426: write_msr: 6e0, value 1d692fed81fe1
<idle>-0     [005] dNh2 259727.776431: local_timer_exit: vector=238
…
<idle>-0     [005] d..3 259727.776482: sched_switch: prev_comm=swapper/5 prev_pid=0
        prev_prio=120 prev_state=S ==> next_comm=cyclictest next_pid=20049 next_prio=19
<idle>-0     [000] d.h3 259727.776499: hrtimer_cancel: hrtimer=00000000f8e18f9f
<idle>-0     [000] d.h2 259727.776505: hrtimer_expire_entry: hrtimer=00000000f8e18f9f
        function=hrtimer_wakeup now=259728721865744
```

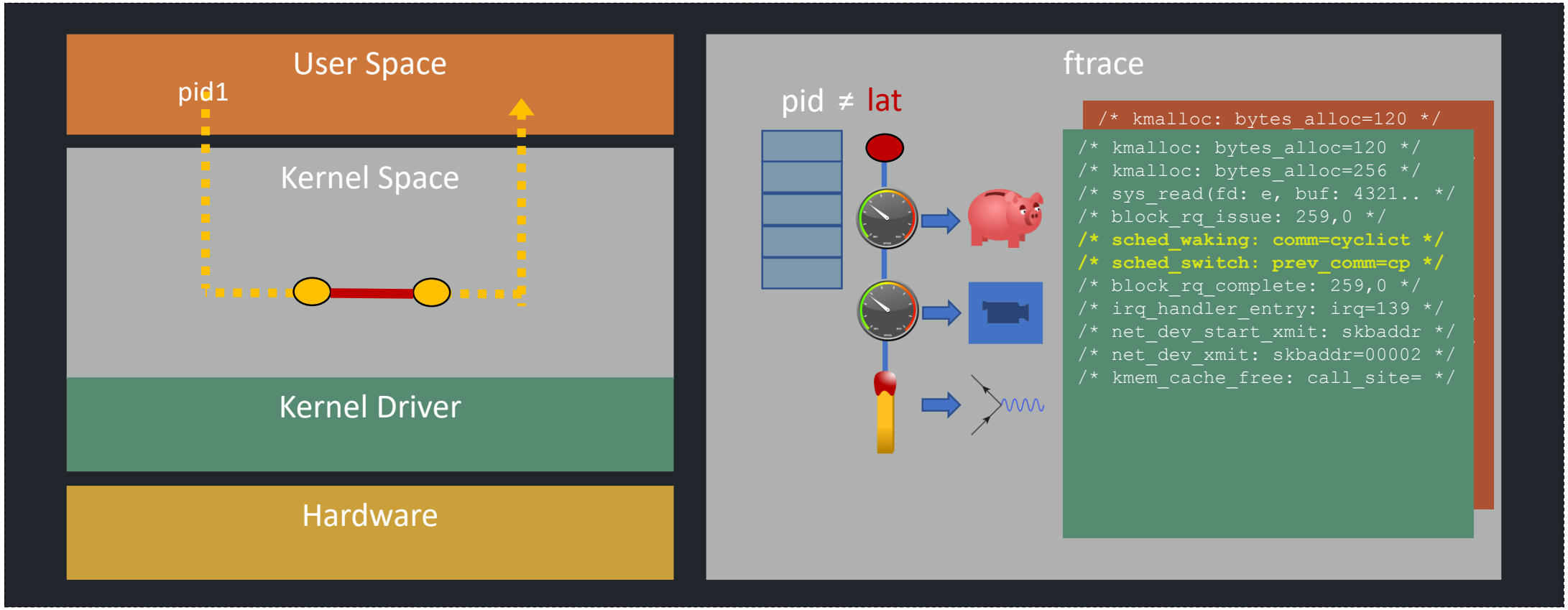
What about a latency histogram?

- We have a latency and would like a histogram of latencies but our histogram is keyed on pid



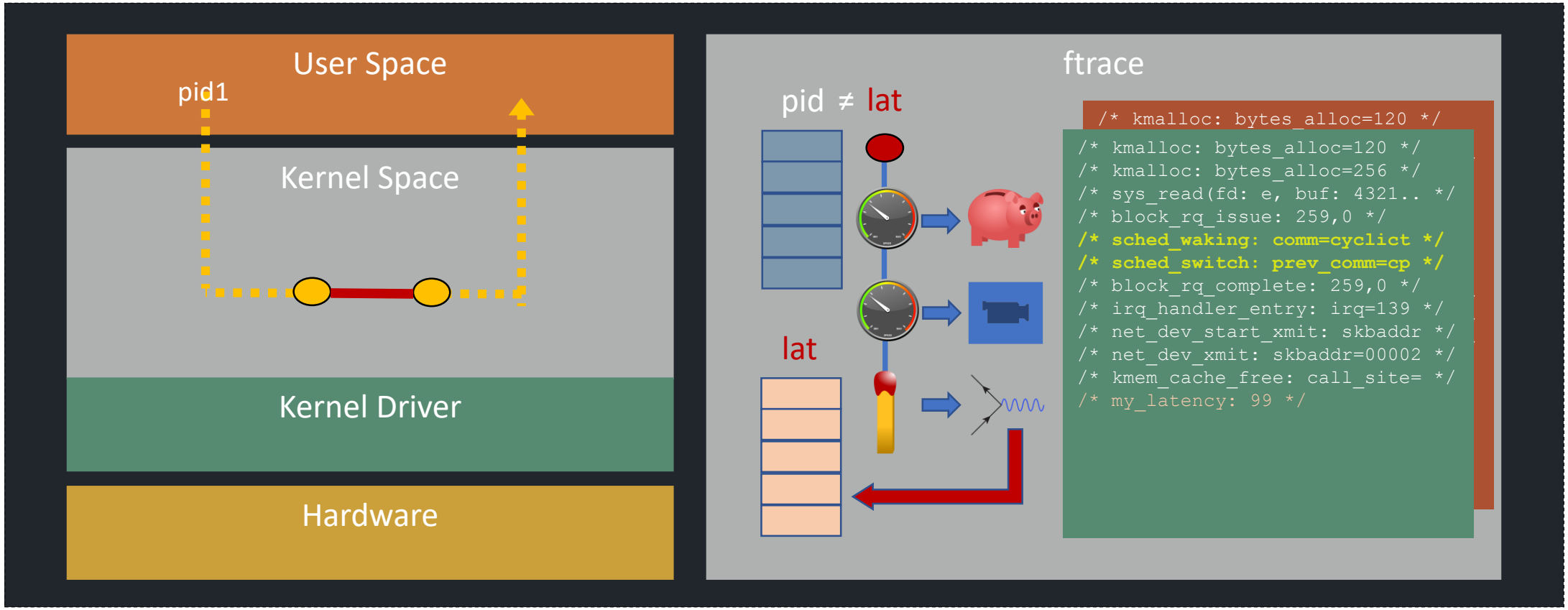
What about a latency histogram?

- Every hit (match) that generates a latency should generate an entry in a latency histogram



What about a latency histogram?

- We can create our own ‘synthetic’ latency event and generate a histogram using that



onmatch synth + onmax save/snapshot

- `onmatch(subsys.event).waking_latency()`
  - `onmatch()` is the 'handler', `waking_latency()` is the 'action' (synthetic event)

```
# echo 'waking_latency u64 lat; pid_t pid' >> /sys/kernel/debug/tracing/synthetic_events

# echo 'hist:keys=pid:ts0=common_timestamp.usecs if comm=="cyclictest" && prio < 100'
        >> /sys/kernel/debug/tracing/events/sched/sched_waking/trigger

# echo 'hist:keys=next_pid:waking_lat=common_timestamp.usecs-$ts0:
        onmax($waking_lat).save(next_comm,next_prio,prev_pid,prev_comm,prev_prio):
        onmatch(sched.sched_waking).waking_latency($waking_lat,next_pid):
        onmax($waking_lat).snapshot()
                if next_comm=="cyclictest"' >> events/sched/sched_switch/trigger

# echo 'hist:keys=pid,lat:sort=pid,lat' >> events/synthetic/waking_latency/trigger

# echo 1 > /sys/kernel/debug/tracing/events/enable
# echo 0 > /sys/kernel/debug/tracing/events/lock/enable
# cyclictest -p 80 -n -s -t 1 -D 2
```

onmatch synth and save/snap output

- `onmax.save()` output:

```
# cat /sys/kernel/debug/tracing/events/sched/sched_switch/hist

# event histogram
#

{ next_pid:       3928 } hitcount:       1962
         max:         97  next_comm: cyclictest                    next_prio:         19
         prev_pid:     0  prev_comm: swapper/2                     prev_prio:        120
         max:         97

Totals:
    Hits: 1962
    Entries: 1
    Dropped: 0
    Deletes: 0
```

onmax and save and snapshot output

- `onmax.snapshot()` output:

```
# cat /sys/kernel/debug/tracing/snapshot

<idle>-0     [002] d.h2  5260.133221: hrtimer_expire_entry: hrtimer=00000000c774a77c
        function=hrtimer_wakeup now=5260158961873
<idle>-0     [002] d.h3  5260.133228: sched_waking: comm=cyclictest pid=3928 prio=19
<idle>-0     [002] dNh4  5260.133259: sched_wakeup: comm=cyclictest pid=3928 prio=19
<idle>-0     [002] dNh2  5260.133264: hrtimer_expire_exit: hrtimer=00000000c774a77c
<idle>-0     [002] dNh2  5260.133272: write_msr: 6e0, value 98d4c3f2192
<idle>-0     [002] dNh2  5260.133275: local_timer_exit: vector=238
<idle>-0     [002] .N.2  5260.133283: cpu_idle: state=4294967295 cpu_id=2
<idle>-0     [002] dN.2  5260.133323: rcu_utilization: End context switch
<idle>-0     [002] d..3  5260.133325: sched_switch: prev_comm=swapper/2 prev_pid=0
        prev_prio=120 prev_state=S ==> next_comm=cyclictest next_pid=3928 next_prio=19
<idle>-0     [002] d..6  5260.133334: waking_latency: lat=97 pid=3928
```

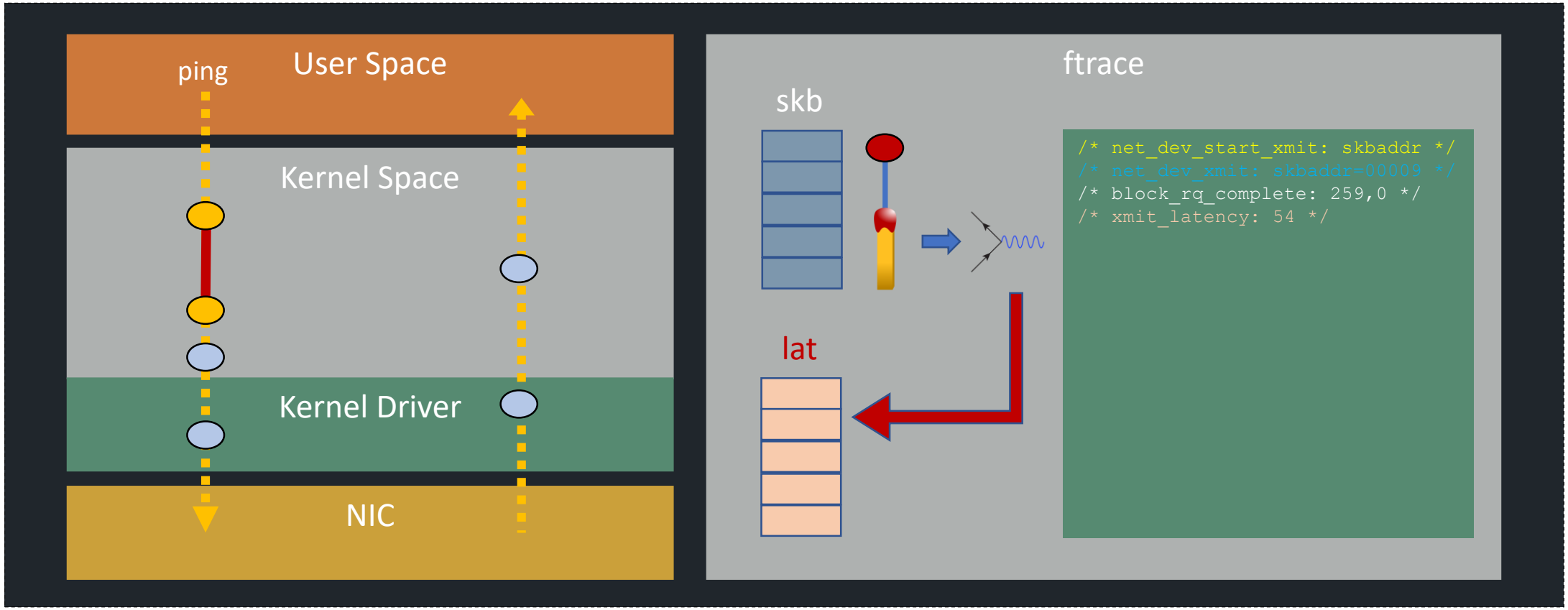
onmax and save and snapshot output

- `waking_latency` histogram:

```
# cat /sys/kernel/debug/tracing/events/synthetic/waking_latency/hist

# event histogram
#

{ pid:       3928, lat:          3 } hitcount:          6
{ pid:       3928, lat:          4 } hitcount:        749
{ pid:       3928, lat:          5 } hitcount:        796
{ pid:       3928, lat:          6 } hitcount:        105
{ pid:       3928, lat:          7 } hitcount:         38
…
{ pid:       3928, lat:         95 } hitcount:          1
{ pid:       3928, lat:         97 } hitcount:          1

Totals:
    Hits: 1962
    Entries: 65
```

Object latencies

- In many cases, we may want to use an object (address) as a histogram key



Object latencies

- We use the `skbaddr` field as a key
  - Essentially this amounts to tracking skb 'objects' through the kernel

```
# echo 'xmit_latency u64 start_xmit_to_xmit_lat; int pid' >>
        /sys/kernel/debug/tracing/synthetic_events

# echo 'hist:tag=xmit:keys=skbaddr.hex:ts0=common_timestamp.usecs' >>
        /sys/kernel/debug/tracing/events/net/net_dev_start_xmit/trigger

# echo 'hist:tag=xmit:keys=skbaddr.hex:start_xmit_to_xmit=common_timestamp.usecs-$ts0:
        onmatch(net.net_dev_start_xmit).xmit_latency($start_xmit_to_xmit,common_pid)'
        >> /sys/kernel/debug/tracing/events/net/net_dev_xmit/trigger

# echo 'hist:keys=pid,start_xmit_to_xmit_lat:sort=pid,start_xmit_to_xmit_lat' >>
        /sys/kernel/debug/tracing/events/synthetic/xmit_latency/trigger
```

Object latencies

- But the tracking histogram quickly gets full because entries aren't discarded after use

```
# cat /sys/kernel/debug/tracing/events/net/net_dev_start_xmit/hist

{ skbaddr: ffff8f3c5e598ed0 } hitcount:         73
{ skbaddr: ffff8f3c5e59ac90 } hitcount:         74
{ skbaddr: ffff8f3c5e5987d0 } hitcount:         75
{ skbaddr: ffff8f3c5e599b10 } hitcount:         79
{ skbaddr: ffff8f3c5e59b390 } hitcount:         87
{ skbaddr: ffff8f3c5e599250 } hitcount:         90
{ skbaddr: ffff8f3c5e598990 } hitcount:         94
{ skbaddr: ffff8f3c5e59a750 } hitcount:         95
{ skbaddr: ffff8f3c5e598450 } hitcount:        105
{ skbaddr: ffff8f3c5e59b8d0 } hitcount:        107

Totals:
    Hits: 7086
    Entries: 2048
    Dropped: 1711
```

tags and delete() (and removeall())

- `tracing_map delete()` operation added by Vedang Patel
- Hist trigger action `delete(tagname)` requires 'tags'
  - Every 'tagged' histogram's matching entry is deleted when invoked
  - The last histogram in a chain should invoke the `delete()` operation
    - Which itself should be tagged
- `removeall(tag)` can be used to remove all triggers with a tag
  - One step toward higher-level 'trigger management'

Object latencies (using delete())

- Label each participating histogram with a tag (`xmit`)

```
# echo 'xmit_latency u64 start_xmit_to_xmit_lat; int pid' >>
        /sys/kernel/debug/tracing/synthetic_events

# echo 'hist:tag=xmit:keys=skbaddr.hex:ts0=common_timestamp.usecs if comm=="ping"'
        >> /sys/kernel/debug/tracing/events/net/net_dev_start_xmit/trigger

# echo 'hist:tag=xmit:keys=skbaddr.hex:start_xmit_to_xmit=common_timestamp.usecs-$ts0:
        onmatch(net.net_dev_start_xmit).xmit_latency($start_xmit_to_xmit,common_pid)
        :delete(xmit) if comm=="ping"' >>
        /sys/kernel/debug/tracing/events/net/net_dev_xmit/trigger

# echo 'hist:keys=common_type:maxlat=start_xmit_to_xmit_lat:onmax($maxlat).snapshot()' >>
        /sys/kernel/debug/tracing/events/synthetic/xmit_latency/trigger

# echo 'hist:removeall(xmit)' >> /sys/kernel/debug/tracing/events/net/net_dev_start_xmit/trigger
```

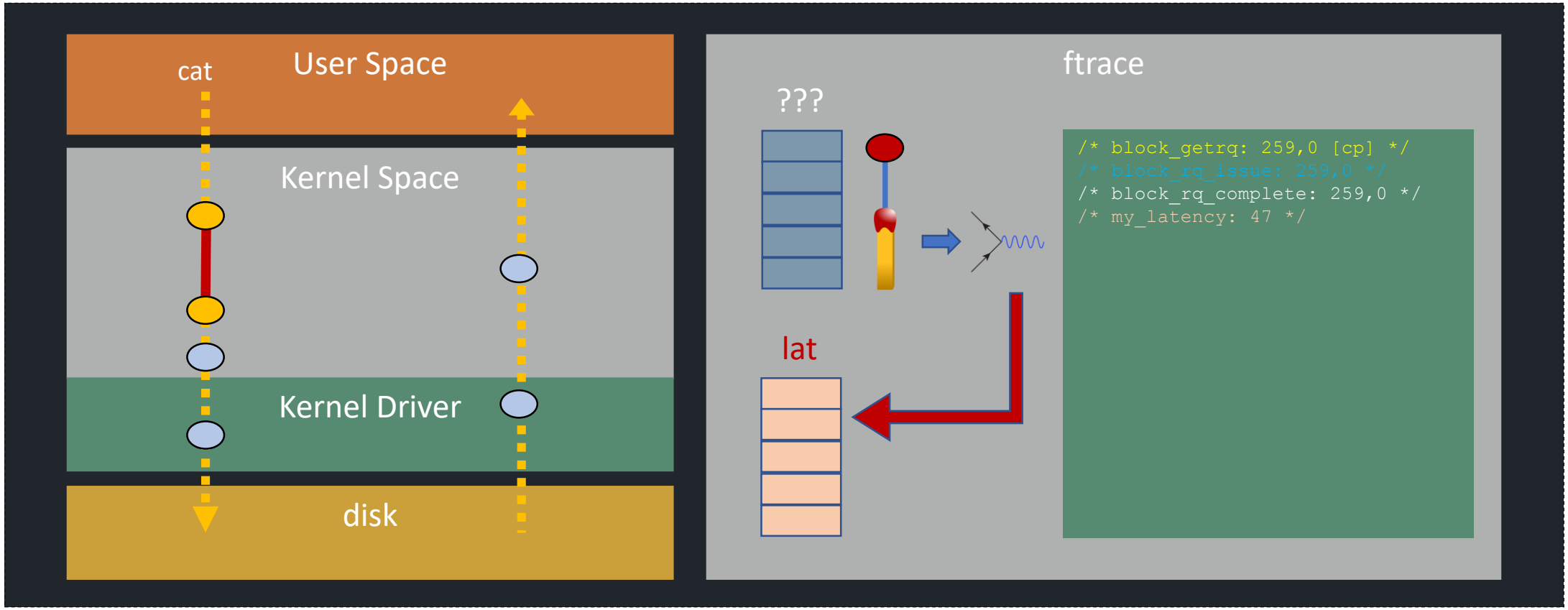
Object latencies (using delete())

- Note that there are no entries in this histogram, and 5 deletes:

```
# cat /sys/kernel/debug/tracing/events/net/net_dev_start_xmit/hist

# event histogram
#

Totals:
    Hits: 5
    Entries: 0
    Dropped: 0
    Deletes: 5
```

Object latencies, part 2

- Suppose we want to look into the block layer to solve a problem



Object latencies, part 2

- Start by looking at block and fs trace events running workload with `trace-me-block`

```
#!/bin/sh

DEBUGFS=`grep debugfs /proc/mounts | awk '{ print $2; }'`
echo $$ > $DEBUGFS/tracing/set_event_pid
echo 'block:* ext4:*' > $DEBUGFS/tracing/set_event
echo 1 > $DEBUGFS/tracing/tracing_on
exec $*
echo 0 > $DEBUGFS/tracing/tracing_on
echo '!block:* !ext4:*' > $DEBUGFS/tracing/set_event
echo > $DEBUGFS/tracing/set_event_pid
```

Object latencies, part 2

- `trace-me-block` output:

```
# echo 3 > /proc/sys/vm/drop_caches;trace-me-block cat junktrace > /dev/null;
        cat /sys/kernel/debug/tracing/trace | less

cat-3524  [003] ...1  9759.086434: ext4_es_insert_extent: dev 259,2 ino 49032859 es [640/64)
cat-3524  [003] ...2  9759.086473: block_bio_remap: 259,0 R 1570544640 + 512 <- (259,2) 156949401 6
cat-3524  [003] ...1  9759.086473: block_bio_queue: 259,0 R 1570544640 + 512 [cat]
cat-3524  [003] ...1  9759.086475: block_split: 259,0 R 1570544640 / 1570544896 [cat]
cat-3524  [003] ...1  9759.086475: block_getrq: 259,0 R 1570544640 + 256 [cat]
cat-3524  [003] ...1  9759.086477: block_getrq: 259,0 R 1570544896 + 256 [cat]
cat-3524  [003] ...2  9759.086479: block_rq_issue: 259,0 R 131072 () 1570544640 + 256 [cat]
cat-3524  [003] ...1  9759.086481: block_unplug: [cat] 1
cat-3524  [003] ...2  9759.086481: block_rq_insert: 259,0 R 131072 () 1570544896 + 256 [cat]
cat-3524  [003] ...3  9759.086483: block_rq_issue: 259,0 R 131072 () 1570544896 + 256 [cat]
…
cat-3524  [003] d.h4  9759.086851: block_rq_complete: 259,0 R () 1570544640 + 256 [0]
cat-3524  [003] d.h4  9759.086853: block_rq_complete: 259,0 R () 1570544896 + 256 [0]
```

Object latencies, part 2

- Existing `block_rq_issue` trace event:

```
# cat /sys/kernel/debug/tracing/events/block/block_rq_issue/format

name: block_rq_issue
ID: 1039
format:
        field:unsigned short common_type;  offset:0;         size:2;  signed:0;
        field:unsigned char common_flags;  offset:2;         size:1;  signed:0;
        field:unsigned char common_preempt_count;  offset:3;         size:1;  signed:0;
        field:int common_pid;      offset:4;         size:4;  signed:1;

        field:dev_t dev;  offset:8;         size:4;  signed:0;
        field:sector_t sector;     offset:16;        size:8;  signed:0;
        field:unsigned int nr_sector;      offset:24;        size:4;  signed:0;
        field:unsigned int bytes;  offset:28;        size:4;  signed:0;
        field:char rwbs[8];        offset:32;        size:8;  signed:1;
        field:char comm[16];       offset:40;        size:16; signed:1;
        field:__data_loc char[] cmd;       offset:56;        size:4;  signed:1;
```

Object latencies, part 2

- Look at events and surrounding functions with `graphtrace-me-block`

```
#!/bin/sh

DEBUGFS=`grep debugfs /proc/mounts | awk '{ print $2; }'`
echo $$ > $DEBUGFS/tracing/set_ftrace_pid
echo $$ > $DEBUGFS/tracing/set_event_pid
echo function_graph > $DEBUGFS/tracing/current_tracer
echo 'block:*' > $DEBUGFS/tracing/set_event
echo 1 > $DEBUGFS/tracing/tracing_on
exec $*
echo 0 > $DEBUGFS/tracing/tracing_on
echo 'block:*' > $DEBUGFS/tracing/set_event
echo > $DEBUGFS/tracing/set_event_pid
```

Object latencies, part 2

- `graphtrace-me-block` output:

```
# echo 3 > /proc/sys/vm/drop_caches;graphtrace-me-block cat junktrace >
        /dev/null;cat /sys/kernel/debug/tracing/trace | less

6)               |                                  blk_mq_try_issue_directly() {
6)               |                                      nvme_queue_rq() {
6)   1.006 us    |                }
6)               |                blk_mq_start_request() {
6)               |                  rcu_read_lock_sched_held() {
6)               |                  /* block_rq_issue: 259,0 R 131072 () 1570544640 + 256 [cat] */
6)   ==========> |
6)               |              do_IRQ() {
6)               |                nvme_irq() {
6)               |                    blk_mq_end_request() {
6)               |                      blk_update_request() {
6)               |                      /* block_rq_complete: 259,0 R () 1570544640 + 256 [0] */
6)   0.141 us    |                      blk_account_io_completion();
6)               |                      bio_endio() {
6)               |                        bio_put() {
```

Function Event Magic

- Here are the function signatures for those two functions:

```
block/blk-mq.c:   void blk_mq_start_request(struct request *rq)
block/blk-core.c: void blk_account_io_completion(struct request *req, unsigned int bytes)
```

- Steve Rostedt's new 'function event' patchset allows them to turn into trace events:

```
# echo 'blk_mq_start_request(x64 rq)' >> /sys/kernel/debug/tracing/function_events
# echo 'blk_account_io_completion(x64 rq)' >> /sys/kernel/debug/tracing/function_events
```

- Similar functionality exists via kprobes (new fetcharg interface by Masami Hiramatsu):

```
# echo p:blk_mq_start_request blk_mq_start_request rq=%bx:x64 >> tracing/kprobe_events
# echo p:blk_account_io_completion blk_account_io_completion req=%di:x64 >> tracing/kprobe_events
```

- And perf-probe:

```
# perf probe -a 'blk_mq_start_request rq'
# perf probe -a 'blk_account_io_completion req'
```

Function Event Magic

- `blk_mq_start_request` function event:

```
# cat /sys/kernel/debug/tracing/events/functions/blk_mq_start_request/format

name: blk_mq_start_request
ID: 1732
format:
        field:unsigned short common_type;  offset:0;        size:2;  signed:0;
        field:unsigned char common_flags;  offset:2;        size:1;  signed:0;
        field:unsigned char common_preempt_count;  offset:3;        size:1;  signed:0;
        field:int common_pid;     offset:4;        size:4;  signed:1;

        field:unsigned long __parent_ip;   offset:8;        size:8;  signed:0;
        field:unsigned long __ip; offset:16;       size:8;  signed:0;
        field:x64 rq;     offset:24;       size:8;  signed:0;

print fmt: "%pS->%pS(rq=%llx)", REC->__ip, REC->__parent_ip, REC->rq
```

Function Event Latency

- `rqlat` histogram definition using our new function events:

```
# echo 'rqlat u64 rqlat; int pid' >> /sys/kernel/debug/tracing/synthetic_events

# echo 'hist:tag=rqlat:keys=rq:ts0=common_timestamp.usecs if comm=="cat"' >>
        /sys/kernel/debug/tracing/events/functions/blk_mq_start_request/trigger

# echo 'hist:tag=rqlat:keys=rq:req_to_complete=common_timestamp.usecs-$ts0:
        onmatch(functions.blk_mq_start_request).rqlat($req_to_complete,common_pid):
        delete(rqlat) if comm=="cat"' >>
        /sys/kernel/debug/tracing/events/functions/blk_account_io_completion/trigger

# echo 'hist:keys=pid,rqlat:sort=pid,rqlat' >>
        /sys/kernel/debug/tracing/events/synthetic/rqlat/trigger
```

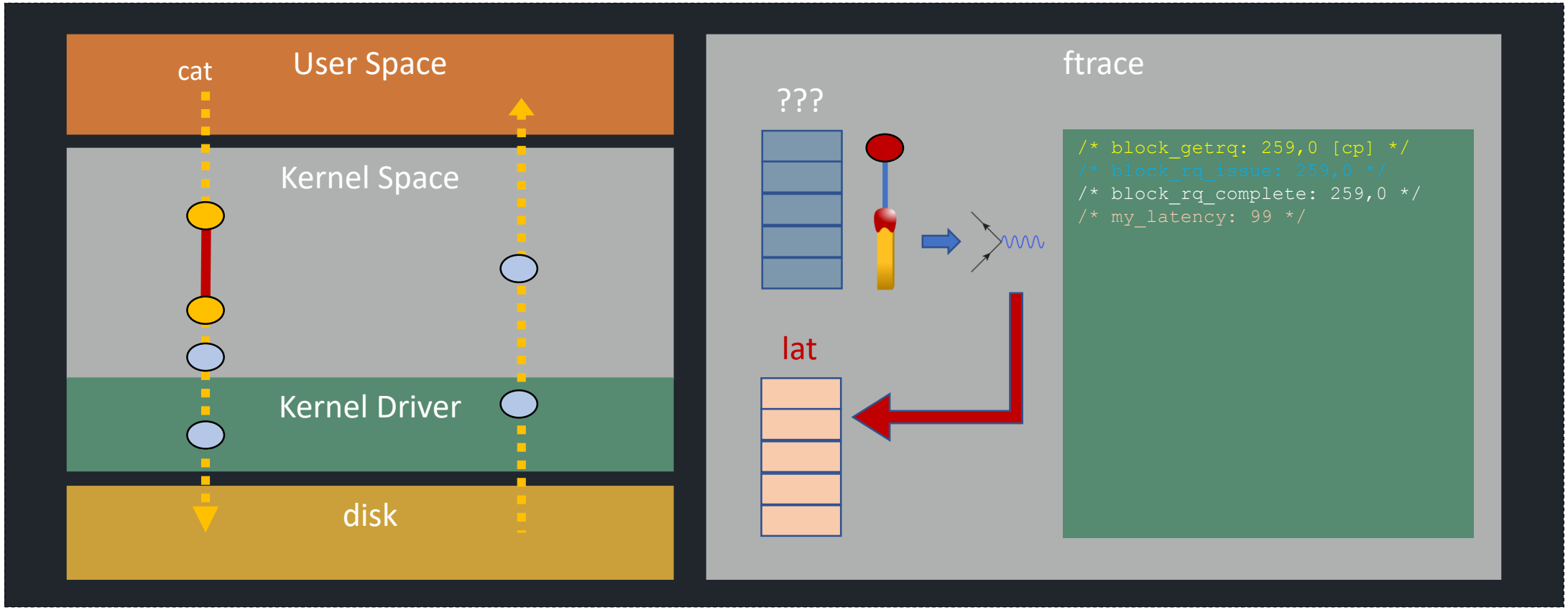
Function Event Latency

- `rqlat` histogram output using our new function events:

```
# cat /sys/kernel/debug/tracing/events/synthetic/rqlat/hist

# event histogram
#

{ pid:       2793, rqlat:         88 } hitcount:          1
{ pid:       2793, rqlat:         89 } hitcount:          1
{ pid:       2793, rqlat:         92 } hitcount:          1
{ pid:       2793, rqlat:        106 } hitcount:          3
…
{ pid:       2793, rqlat:        814 } hitcount:          1
{ pid:       2793, rqlat:        878 } hitcount:          1
{ pid:       2793, rqlat:        937 } hitcount:          1
{ pid:       2793, rqlat:        994 } hitcount:          1

Totals:
    Hits: 69
    Entries: 51
```

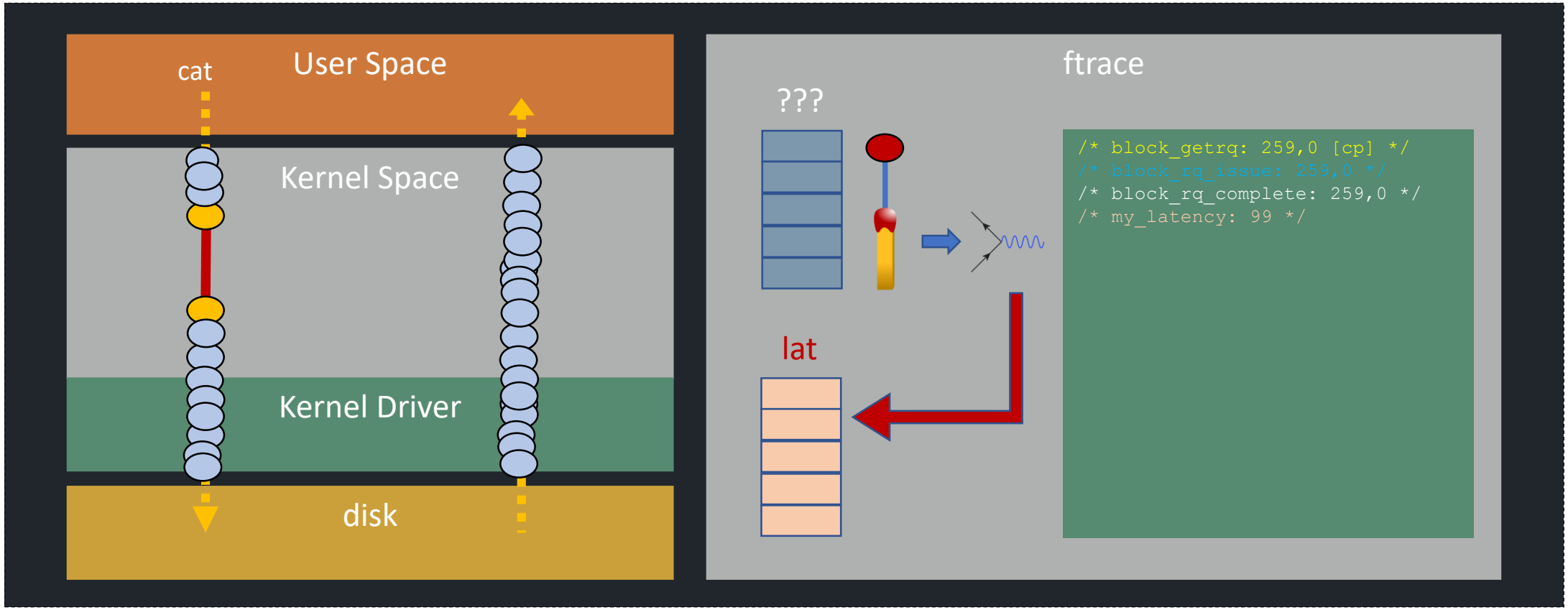
Function Event Magic

- Allows us to go from this:



Function Event Magic

- To this:



Generate latency hist for any 2 events

- usage: `funcevent-latency func1 func2 key [log2]`

```
EVENT1=$1
EVENT2=$2
KEY=$3

echo "${EVENT1}(x64 ${KEY})" >> /sys/kernel/debug/tracing/function_events
echo "${EVENT2}(x64 ${KEY})" >> /sys/kernel/debug/tracing/function_events

echo "funclat u64 lat; int pid" >> /sys/kernel/debug/tracing/synthetic_events

echo "hist:tag=funclat:keys=${KEY}:ts0=common_timestamp.usecs" >>
        /sys/kernel/debug/tracing/events/functions/${EVENT1}/trigger

echo "hist:tag=funclat:keys=${KEY}:lat=common_timestamp.usecs-\$ts0:
        onmatch(functions.${EVENT1}).funclat(\$lat,common_pid):delete(funclat)" >>
        /sys/kernel/debug/tracing/events/functions/${EVENT2}/trigger

echo "hist:keys=pid,lat.log2:sort=pid,lat" >>
        /sys/kernel/debug/tracing/events/synthetic/funclat/trigger
```

General purpose latency script output

- `funclat` histogram output using the `funcevent-latency` script

```
# funcevent-latency blk_mq_start_request blk_account_io_completion rq log2

# cat /sys/kernel/debug/tracing/events/synthetic/funclat/hist

# event histogram

{ pid:       3847, lat: ~ 2^6  } hitcount:          1
{ pid:       4146, lat: ~ 2^5  } hitcount:         36
{ pid:       4146, lat: ~ 2^6  } hitcount:         17
{ pid:       4146, lat: ~ 2^9  } hitcount:          4
{ pid:       4216, lat: ~ 2^5  } hitcount:         21
{ pid:       4216, lat: ~ 2^6  } hitcount:         32
{ pid:       4216, lat: ~ 2^8  } hitcount:          7
{ pid:       4216, lat: ~ 2^9  } hitcount:          6

Totals:
    Hits: 214
    Entries: 19
```

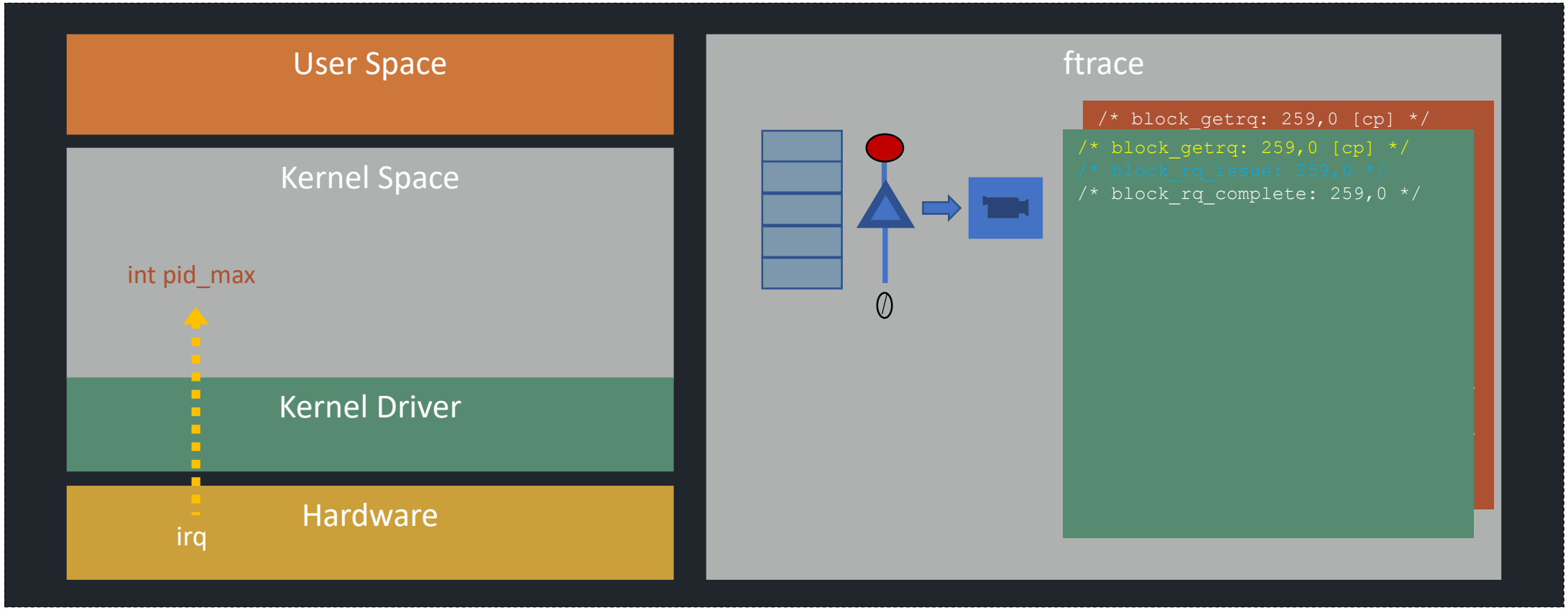
Questions?

- Code at `https://github.com/tzanussi/linux-trace-inter-event/tree/tzanussi/elc-2018`
- Clip art from http://clipart-library.com/
- Thanks to the Linux Foundation for funding my trip.

Backup

onchange() with global variable

- Every irq checks global `pid_max`. Whenever `pid_max` changes, take a snapshot

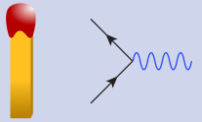
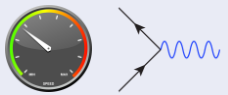
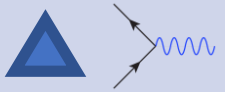


onchange() with global variable

- `pid_max` is a global kernel variable made accessible via a function event on `do_IRQ()`
  - On every irq, an event with the value of `pid_max` is generated
- Whenever that value changes, the new `onchange()` handler takes a snapshot

```
# echo 'do_IRQ(int $pid_max)' > /sys/kernel/debug/tracing/function_events

# echo 'pidmaxchange int pid_max' >> /sys/kernel/debug/tracing/synthetic_events

# echo 1 > /sys/kernel/debug/tracing/events/enable
# echo 0 > /sys/kernel/debug/tracing/events/lock/enable

# echo 'hist:keys=common_type:onmatch(functions.do_IRQ).pidmaxchange(pid_max):
        pidmax=pid_max:onchange($pidmax).snapshot()'
        >> /sys/kernel/debug/tracing/events/functions/do_IRQ/trigger

# echo 65537 > /proc/sys/kernel/pid_max
```

Handlers and Actions Summary

	save	snapshot	generate synthetic event
onmatch			
onmax			
onchange			

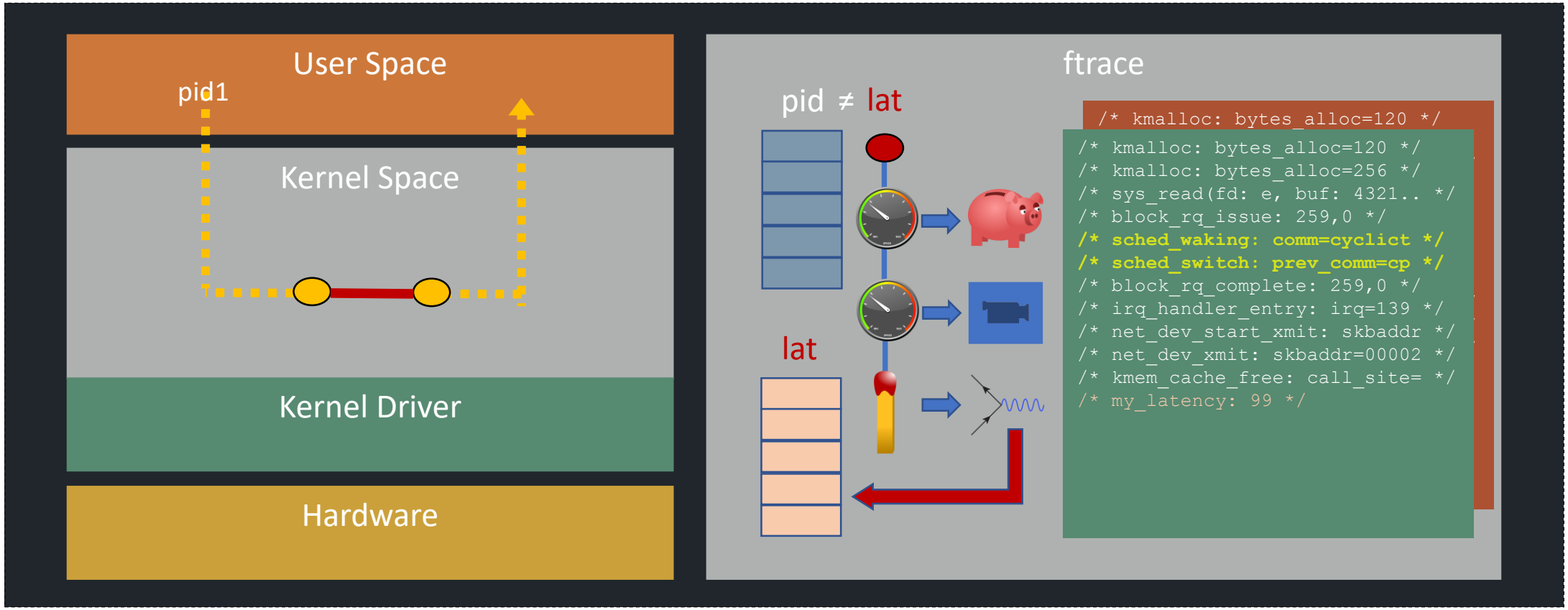
Latency

- What we just did is the equivalent of this code:
  - Save a value in a slot, retrieve it later and replace it with something else:

```
event sched_wakeup()
{
    timestamp[wakeup_pid] = now();
}

event sched_switch()
{
    if (timestamp[next_pid])
        latency = now() - timestamp[next_pid] /* next_pid == wakeup_pid */
        wakeup_latency[next_pid] = latency
        timestamp[next_pid] = null
}
```

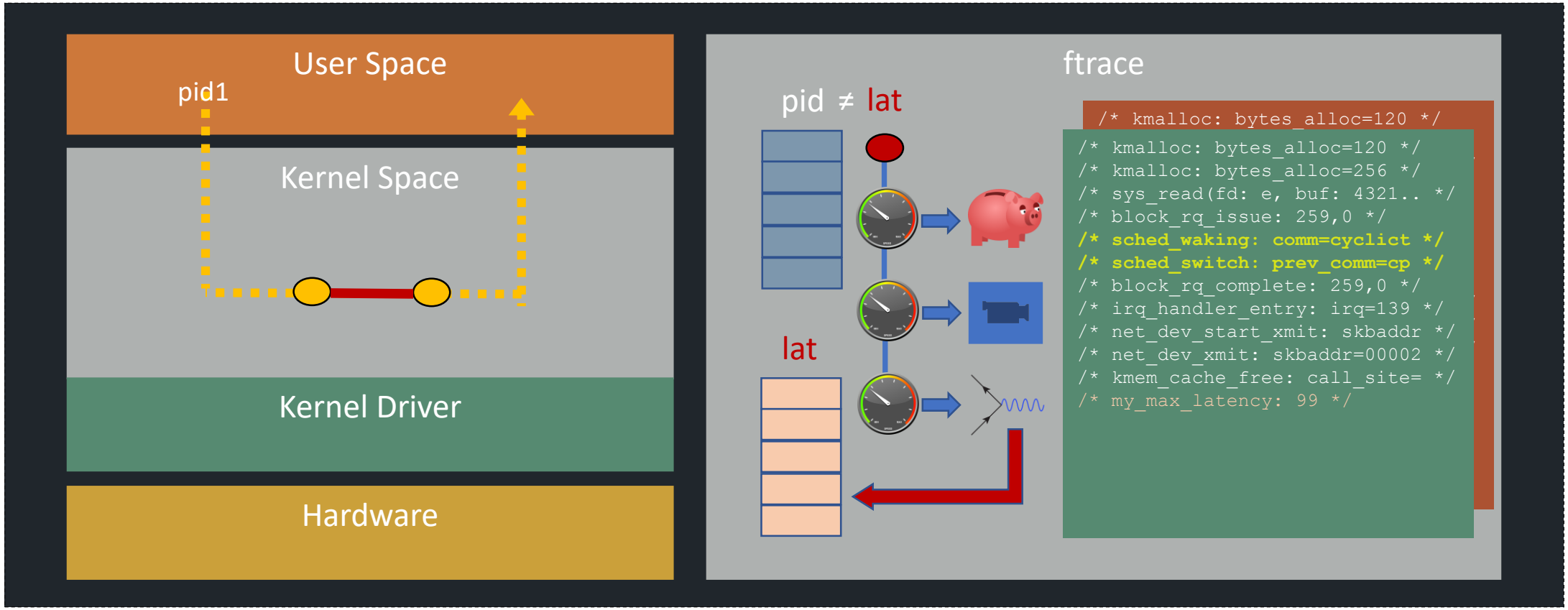
What about a latency histogram?

- We could also replace `onmatch()` with `onmax()` and generate an event only on max



What about a latency histogram?

- Simply replace `onmatch()` with `onmax()` in the synthetic event generating code



onmax.tracexxx()

- `onmax().max_waking_latency(…):onmax().snapshot()` output:

```
# cat /sys/kernel/debug/tracing/snapshot

<idle>-0     [002] d.h2  5260.133221: hrtimer_expire_entry: hrtimer=00000000c774a77c
        function=hrtimer_wakeup now=5260158961873
<idle>-0     [002] d.h3  5260.133228: sched_waking: comm=cyclictest pid=3928 prio=19
<idle>-0     [002] dNh4  5260.133259: sched_wakeup: comm=cyclictest pid=3928 prio=19
<idle>-0     [002] dNh2  5260.133264: hrtimer_expire_exit: hrtimer=00000000c774a77c
<idle>-0     [002] dNh2  5260.133272: write_msr: 6e0, value 98d4c3f2192
<idle>-0     [002] dNh2  5260.133275: local_timer_exit: vector=238
<idle>-0     [002] .N.2  5260.133283: cpu_idle: state=4294967295 cpu_id=2
<idle>-0     [002] dN.2  5260.133323: rcu_utilization: End context switch
<idle>-0     [002] d..3  5260.133325: sched_switch: prev_comm=swapper/2 prev_pid=0
        prev_prio=120 prev_state=S ==> next_comm=cyclictest next_pid=3928 next_prio=19
<idle>-0     [002] d..6  5260.133339: max_waking_latency: lat=97 pid=3928
```

max latency histogram, stacktrace as key

- echo 'hist:keys=**stacktrace**,lat:sort=lat' >> .../max_waking_latency:

```
# cat /sys/kernel/debug/tracing/events/synthetic/max_waking_latency/hist

{ stacktrace:
         futex_wait_queue_me+0xd0/0x160
         futex_wait+0xeb/0x240
         do_futex+0x47f/0xaf0
         SyS_futex+0x12d/0x180
         entry_SYSCALL_64_fastpath+0x29/0xa0
, lat:          474} hitcount:          1
{ stacktrace:
         SyS_epoll_wait+0xcf/0xf0
         do_syscall_64+0x64/0x1c0
         return_from_SYSCALL_64+0x0/0x75
, lat:         3046} hitcount:          1

Totals:
    Hits: 27
    Entries: 27
```

Normal Events

- In kernel code, `trace_xxx()` is used to trace an event e.g. `trace_sched_switch()`
  - `__DO_TRACE()`, in `tracepoint.h`, is called by `trace_xxx()`
  - Only if the event has been enabled is the tracepoint function `tp_func.func()` called
- `DECLARE_EVENT` macros create `trace_event_class event_class_xxx` instances
  - They also create `trace_event_raw_event_xxx()` probes and assign to `class.probe()`
- When an event is enabled, `class.reg()` is called, which is `trace_event_reg()`
  - Which calls `tracepoint_probe_register()` with `class.tp and class.probe()`
  - Which sets `tp_func.func` to the probe function e.g. `trace_event_raw_event_xxx()`
- So `trace_xxx()` invokes the enabled `tp_func.func()` which logs the event
  - `trace_event_buffer_reserve`() reserves the space and grabs the timestamp
  - `trace_event_buffer_commit()` calls all the event triggers on the event file

Synthetic Events

- a `synth_event` is basically a user-defined event name and a set of fields
- synthetic event creation calls `register_synth_event()` to create an event_class
  - Instead of `DECLARE_EVENT` macros creating an `event_class_synth`
  - Creates tracepoint in subsys "synthetic" and creates fields using `trace_define_field()`
  - Also creates a `trace_event_raw_event_synth()` probe and assigns to `class.probe()`
- `event_hist_trigger()` is the event trigger used to create a histogram on an event
  - invoked when the event fires, calls `hist_trigger_actions()`
  - `hist_trigger_actions()` calls the action generating a synthetic event, `action_trace()`
  - Which calls `trace_synth()`, which is just like `trace_xxx()` but for synthetic events
  - `trace_synth()` calls the `class.probe()` function `trace_event_raw_event_synth()`
  - `trace_event_buffer_reserve()` reserves and populates the synthetic event
  - `trace_event_buffer_commit()` finishes it off and closes the cycle